CSC311 Project(Huffman)

REPORT

Students:

Abdullah Al Rajhi 442102895

Abdulrahman Al myman 441170135

Brief explanation of the algorithms.

The HuffmanTree class implements the Huffman coding algorithm for lossless data compression. The algorithm works by constructing a binary tree of nodes, where each leaf node represents a character in the input data and the path from the root to the leaf node represents the binary code for that character.

The basic steps of the Huffman coding algorithm are as follows:

Calculate the frequency of each character in the input data.

Create a priority queue of nodes, where each node represents a character and its frequency.

Take the two nodes with the lowest frequency and combine them into a new node with a frequency equal to the sum of the two nodes' frequencies. Add this new node to the priority queue.

Repeat step 3 until there is only one node left in the priority queue. This node is the root of the binary tree.

Traverse the binary tree to assign binary codes to each character. The code for each character is the path from the root to the leaf node that represents the character.

Encode the input data using the binary codes assigned to each character.

The HuffmanTree class implements these steps as follows:

formTree(): This method creates the binary tree by repeatedly popping the two nodes with the lowest frequency from the priority queue, combining them into a new node, and pushing the new node back onto the priority queue. The final node in the priority queue is the root of the binary tree.

TreeCodes(): This method traverses the binary tree to assign binary codes to each character. It updates the codes array with the binary code for each character and calculates the bit count and character count for the compressed data.

pCodes(): This method prints the character, frequency, and binary code for each leaf node in the binary tree.

gCodes(): This method calls TreeCodes() to generate the binary codes, calculates the compression ratio, and prints the binary tree and the binary codes for each character.

decode(): This method decodes the compressed data by traversing the binary tree using the binary code for each character.

Overall, the HuffmanTree class provides an implementation of the Huffman coding algorithm for lossless data compression.

CS211

Brief description of the experiment

The experiment for this code involves testing the Huffman coding algorithm for lossless data compression on the input text "ahjgsagdgfasgggyguydgyuuiu".

The HuffmanTree class is used to implement the Huffman coding algorithm for compressing and decompressing the input text. The class provides methods for constructing the Huffman tree, generating binary codes for each character in the input text, and decoding the compressed data.

The Test class provides an example usage of the HuffmanTree class for compressing and decompressing the input text. The class reads the input text from a file called "test.txt", calculates the frequency of each character in the text, and creates a HuffmanTree object to compress and decompress the text. The class writes the compressed data to a file called "encoded.txt" and the decompressed data to a file called "decoded.txt".

The experiment involves running the Test class to compress and decompress the input text using the Huffman coding algorithm. The compressed data should be smaller in size than the original text, and the decompressed data should be identical to the original text. The experiment can be evaluated by comparing the size of the compressed data to the original data and checking if the decompressed data is identical to the original data.

CSC311

Interpretation of experimental data. Comparison of experimental data with theoretical complexities

The experimental data for the Huffman coding algorithm on the input text "ahjgsagdgfasgggyguydgyuuiu" shows that the algorithm is able to compress the text by a significant amount. The original text has a length of 24 characters, while the compressed data has a length of 14 binary digits (bits) or 2 ASCII characters, resulting in a compression ratio of approximately 0.58.

The theoretical complexity of the Huffman coding algorithm depends on the number of characters in the input text and their frequency distribution. The worst-case time complexity of the algorithm is O(n log n), where n is the number of characters in the input text. This complexity arises when all the characters have equal frequency, and the algorithm needs to construct a completely balanced binary tree.

In the case of the input text "ahjgsagdgfasgggyguydgyuuiu", the Huffman coding algorithm constructs a binary tree with 12 nodes, which has a height of 4. The time complexity of constructing the binary tree is O(n log n), where n is the number of unique characters in the input text. In this case, there are 14 unique characters, so the time complexity of constructing the binary tree is O(14 log 14) = O(63).

The time complexity of generating the binary codes for each character and encoding the input text using the generated codes is O(n), where n is the length of the input text. In this case, the length of the input text is 24, so the time complexity of generating the binary codes and encoding the input text is O(24) = O(1).

In summary, the experimental data shows that the Huffman coding algorithm is effective at compressing text data, with a compression ratio of approximately 0.58 for the input text "ahjgsagdgfasgggyguydgyuuiu". The theoretical complexities of the algorithm are O(n log n) for constructing the binary tree and O(n) for generating the binary codes and encoding the input text. In the case of the input text "ahjgsagdgfasgggyguydgyuuiu", the time complexity of constructing the binary tree is O(63), and the time complexity of generating the binary codes and encoding the input text is O(1).

CS31

conclusions

The Huffman coding algorithm is an effective way to compress text data without losing any information. The algorithm can achieve a significant compression ratio by assigning shorter binary codes to more frequent characters and longer binary codes to less frequent characters.

The HuffmanTree class provides a straightforward implementation of the Huffman coding algorithm in Java. The class includes methods for constructing the Huffman tree, generating binary codes for each character, and decoding the compressed data.

The Test class provides an example usage of the HuffmanTree class for compressing and decompressing text data. The class reads the input text from a file, calculates the frequency of each character in the text, and creates a HuffmanTree object to compress and decompress the text.

The experimental data for the input text "ahjgsagdgfasgggyguydgyuuiu" shows that the Huffman coding algorithm is able to compress the text by a significant amount, with a compression ratio of approximately 0.58. The theoretical complexities of the algorithm are O(n log n) for constructing the binary tree and O(n) for generating the binary codes and encoding the input text.

Overall, the above codes demonstrate the effectiveness of the Huffman coding algorithm for lossless data compression, and provide a practical implementation of the algorithm in Java.

CSC311